INDENICA Variability Modeling Language: Language Specification

Version 1.20

(corresponds to IVML bundle version 0.9.0)

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

Creating domain-specific service platforms requires the capability of customizing and configuring service platforms according to the specific needs of a domain. In this document we address this demand from the perspective of variability modeling. We focus on how to describe customization and configuration options in service (platform) ecosystems using a variability modelling language.

In this document we specify the concepts of the INDENICA variability modelling language (IVML) to describe customization and configuration options in service (platform) ecosystems.

Version

1.0	15. February 2012	first version derived from D2.1
1.01	29. February 2012	“v” as prefix in version number (technical reasons)
1.02	15. June 2012	all parenthesis follow after all “with” keywords, accessing enum literals, camelcase for refTo and refBy
1.03	17. July 2012	DSL syntax corrected
1.04	19. July 2012	Constraint syntax, operation signatures and semantics, grammar section (prepared), technical section, import conventions, clarifications in using constraints
1.05	22. July 2012	grammar revised and added to document
1.06	06. August 2012	typeSelect, typeReject, side effects, undefined values
1.07	10. August 2012	examples testcased, enum access corrected in examples, container type definition adjusted (syntax overlap with variable declaration)
1.08	16. August 2012	Details for identifiers added, technical section deleted (see IVML user guide)
1.09	28. November 2012	Operator precedence in grammar corrected, constraint type added
1.10	12. December 2012	Mass assignment of attribute values
1.11	8. January 2013	Assignment operator clarifications (‘=’ vs. ‘==’)
1.12	17. January 2013	Eval clarified
1.13	11. February 2013	Compound / container initializers and attributes clarified
1.14	12. February 2013	Grammar cleanup, decision variable naming corrected
1.15	14. February 2013	Further clarifications on ‘=’
1.16	10. July 2013	Qualification clarification (reasoner dependent)
1.17	16. August 2013	Clarification on attributes, null values introduced
1.19	4. December 2013	Refined hierarchical import path.
1.20	21. April 2014	Implementation status

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Figures

1 Introduction

This document specifies the INDENICA variability modelling language (IVML) in terms of a live document containing the current version based on discussions with the partners and experiences made during the project.

2 The INDENICA Variability Modelling Approach

In this section, we will describe the concepts of the INDENICA Variability Modelling Language (IVML). In accordance to the previous sections, we will distinguish between a core modelling language and an advanced modelling language that extends the core language to satisfy the specific requirements that arise in the INDENICA project. This distinction facilitates ease of use for the most standard issues in variability modelling as it does not complicate the use of this language for users who do not need the more advanced features. The concepts of the core modelling language are based on the results of the discussion in D2.1. In this section, we discussed different levels of expressiveness for basic variability modelling in INDENICA. The core modelling language is extended by advanced modelling concepts that we identified as prerequisites to effective and efficient variability modelling in service-based systems and, in particular, in service (platform) ecosystems in D2.1.

The basic concepts of the IVML are related to approaches like the Text-based Variability Language (TVL) [2], the Class Feature Relationships (Clafer) [1], the Compositional Variability Management framework (CVM) [7], etc. However, we decided to develop a different approach, based on decision modelling concepts, in order to appropriately address the requirements identified in D2.1.

We will introduce a textual specification to describe the IVML concepts. This will help to give a precise representation of the modelling concepts. The syntax, we use in this section was developed as a basis for representing the concepts. Our presentation of the IVML-syntax draws upon typical concepts used in programming languages, in particular Java, and other modelling languages such as TVL [2], Clafer [1], the Object Constraint Language (OCL) [4], or the UML [5]. The dependency management concepts of the IVML mostly rely on the concepts of the OCL. We will adapt these concepts as needed to provide additional operations required by IVML-specific modelling elements, e.g. match and substitute operations for decision variables of type string.

We will use the following styles and elements throughout this section to illustrate the concepts of the IVML:

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.
- *`Elements and expressions`* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number. We recommend that the identifiers of new types start with a capital letter to easily distinguish them from variables.
- Expressions will be separated using semicolon "`;`".

- Different types of brackets will be used to indicate lists "`()`", sets "`{}`", etc. This is closely related to the Java programming language.
- We will indicate comments using "`//`" and "`/* ... */`" (cf. Java).

We will use the following structure to describe the different concepts:

- **Syntax:** this is the syntax of a concept. We will use this syntax to illustrate the valid definition of elements as well as their combination.
- **Description of syntax:** provides the description of the syntax and the associated semantics. We will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

In Section 2.1, we will describe the INDENICA variability modelling core language. We will introduce the required elements and expressions to define a basic configuration space including Boolean and non-Boolean variabilities. We will further describe the dependency management capabilities of this language to restrict configuration spaces. Finally, we will describe the definition of (product) configurations based on configuration spaces.

In Section 2.2 we will describe the advanced concepts of the INDENICA variability modelling language. We will introduce extensions that are required to satisfy the specific requirements in the INDENICA project like the support for service-ecosystems, for service technology and meta-variability.

2.1 INDENICA Variability Modelling Core Language

This section describes the core language of the IVML. In this language, a project is the top-level element that identifies the configuration space of a certain (software) project. In terms of a product line, this may either be an infrastructure as a basis for deriving products or a final product. In a project the relevant modelling elements will be defined. We describe this in the first part of this section. In the second part, we introduce the type system supported by the IVML. These types can be used to declare different types of decision variables. The dependency management capabilities to restrict the configuration space of a project will be described next. Finally, we will introduce the configuration concept of the IVML, which enables the definition of specific (product) configurations based on the configuration space defined in a project.

2.1.1 Projects

In the IVML a project (**`project`**) is the top-level element in each model. This element is mandatory as it identifies the configuration space of a certain software project and, thus, scopes all variabilities of that software project. The definition of a project requires a name, which simultaneously defines a namespace for all elements of this project.

Syntax:

```
project name {
```

```
    /* Definition of the configuration space and
    configurations. */

}
```

Description of syntax: the definition of a new project consists of the following elements:

- The keyword **`project`** defines that the identifier *`name`* is defined as a new project or, to be more precise, as a new configuration space.
- *`name`* is an identifier that defines the name of the new project and, thus, the namespace of all elements within this project.
- The elements surrounded by curly brackets define the configuration space of the new project.

Example:

```
project contentSharing {

    /* This will define a new project for a content-sharing
    project. This is related to our running example in D2.1
    */

}
```

2.1.2 Types

In a project (cf. Section 2.1.1) different kinds of core modelling elements may be used to both represent the variabilities and define a configuration space appropriately. We will express these kinds as formal types in IVML, thus defining a (strongly) typed language. We distinguish between basic types, enumerations, container types, derived and restricted types and compound types. These types can be used to declare or define concrete decision variables. Basically, all decision variables can be unset using the null keyword, i.e., explicitly assigning no value to a variable.

2.1.2.1 Basic Types

In D2.1, we argued that non-Boolean variability is a must for the core expressiveness of the INDENICA language. Thus, the IVML supports as basic types Boolean (**`Boolean`**), integer (**`Integer`**), real (**`Real`**) and string (**`String`**) with their usual meaning. The names of the basic types are aligned to OCL [4]. These types support the definition of basic variabilities, e.g. the `Boolean` type may be used for modelling optional variabilities. In addition, types like `Integer` or `Real` provide a basis for defining advanced variabilities, e.g. using an `Integer` to define a quantitative property for Quality of Service (QoS) as described in D2.1. In addition, IVML provides the basic type `Constraint` which allows declaring constraints themselves as variable.

2.1.2.2 Enumerations

Enumerations allow the definition of sets of named values. This is used to describe a set of possible resolutions of a decision.

Syntax:

```
enum Name1 {value1, ..., valuen};

enum Name2 {value1=n1, ..., valuen=nn};
```

Description of syntax: the definition of a new enumeration type consists of the following elements:

- The keyword **enum** defines that the identifier *Name* is defined as a new enumeration.
- *Name* is an identifier and defines the name of the new type.
- The identifiers surrounded by curly brackets are the concrete elements of the enumeration. A specific element of an enumeration can be accessed using the "."-notation, e.g. $Name_1$.$value_1$.
- Specifying concrete numeric values for elements of an enumeration ($value_i$=n_i) turns the enumeration into an ordered enumeration. This enables relations like greater than (>) or less than (<) and operations like next (**next**) or previous (**previous**) on the values to be used.

Example:

```
enum Colors {green, yellow, black, white};

enum BindingTimes {configuration=0, compile=1,
  runtime=2};
```

2.1.2.3 Container Types

The IVML provides two container types, sequences and sets. Sequences can contain an arbitrary number of elements of a given content type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. These types can be used to describe a number of possible options out of which several can be selected at the same time. Elements in a container (both sequences and sets) can be accessed by their position in the container using an index (`[index]`). The allowed number of elements in a container, i.e., its cardinality, can be restricted by constraints.

The IVML supports a set of operations specific for container types, e.g. adding or appending elements to a container, deleting elements of a container, selecting specific elements, etc. We will introduce the full set of operations in Section 2.1.4.

Syntax:

```
// Declaration of a new sequence and a new set.
```

```
sequenceOf(Type) variableName1;

setOf(Type) variableName2;


/* Access to elements of a sequence. Sets do not have
index-based access. We will discuss variables in Section
2.1.3. */

variableName1[index] = value;
```

Description of Syntax: the definition of a container type consists of the following elements:

- The **`sequenceOf`** and **`setOf`** keywords refer to a container of the respective type followed by the *`Type`* of the elements contained in brackets.
- The identifiers *`Name`*$_1$ and *`Name`*$_2$ are the names of the new containers.
- Accessing a specific element of a sequence container type (variable) requires the specification of an index (`[`*`index`*`]`). An index is either "0" or a positive integer value specifying the position of an element in a container. Accessing a specific position is only a valid operation, if this position has previously been set by different means like the **`add`** function (the set of operations is introduced in Section 2.1.4).

Example:

```
/* Definition of a new enumeration. "blob" means "binary
(large) objects". */

enum ContentType {text, video, audio, threeD, blob};


/* Denotes types of contents supported by a system */

sequenceOf(ContentType) basicContents =
    {ContentType.text, ContentType.audio};
```

2.1.2.4 Type Derivation and Restriction

The IVML allows the derivation of new types based on existing types. This supports extensibility and adaptability as users may define their own types based on basic types, enumerations or container types as well as on previously derived types. The derivation may also include restrictions to the existing type, e.g. to restrict the possible values of the new type to a subset of the values of the existing type. The restrictions are defined by one or more constraints (we will discuss constraints in detail below). Multiple constraints are implicitly combined by a Boolean OR. Thus, at least one constraint has to be satisfied by the new type. The constraints will be defined in OCL style as described in Section 2.1.4.

Syntax:

```
typedef Name1 Type;

typedef Name2 Type with (constraint1, ...,
  constraintn);
```

Description of Syntax: the definition of a derived type consists of the following elements:

- The **typedef** keyword indicates the derivation of a new type based on an existing type.
- The identifiers *$Name_1$* and *$Name_2$* are the names of the new types.
- The identifier *Type* denotes the basic type from which the new type (*$Name_1$* or *$Name_2$*) will be derived.
- The optional keyword **with** introduces a non-empty set of constraints (c.f. Section 2.1.4), surrounded by brackets, out of which at least one must hold for *$Name_2$*. In case of deriving *$Name_2$* from **String** the constraints may define regular expressions.

Example:

```
/* Definition of a type "AllowedBitrates" which is a set
of Integers, i.e. a kind of alias for a complex type
definition. */

typedef AllowedBitrates setOf(Integer);


/* A new modelling type of the basic type integer that is
restricted to assume values between "128" and "256". */

typedef Bitrate Integer with (Bitrate >= 128 and
  Bitrate <= 256);
```

2.1.2.5 Compounds

A compound type groups multiple types into a single named unit (similar to structs or records in programming languages or groups in feature modelling). This allows combining semantically related decisions from which each element has to be configured individually.

Syntax:

```
compound Name {

  Type name1;
```

```
  ...
}
```

Description of Syntax: the definition of a compound type consists of the following elements:

- The **`compound`** keyword indicates the definition of a new compound type.
- The identifier *`Name`* defines the name of the new compound type.
- The set of elements surrounded by curly brackets defines the types of the compound type. Each declaration of a typed element is separated by a semicolon.

Example:

```
/* A new compound type for the configuration of different (web) content. The content may vary in terms of name and bitrate. "Content.bitrate" is the integer within the compound content. */

compound Content {

  String name;

  Integer bitrate;

}
```

2.1.3 Decision Variables

The types introduced in Section 2.1.2 can be used to declare (decision) variables representing a concrete variability. A decision variable is an element of a project (configuration space) that basically accepts any value of its type. Constraints may further restrict the possible values by removing certain combinations of values from the allowed configuration space. The value given to a decision variable defines the variant of the represented variability.

In IVML a decision variable may either be declared with or without a default value (this is an optional parameter). Decision variables with a default value can be further configured by overwriting their (default) value at a later point in time. However, overwriting the default value is not necessary.

Syntax:

```
// Declaration of a decision variable.

Type name1;


/* Declaration of a decision variable with a default value. The "valueAssignment"-expression will be described in detail below. */
```

```
Type name2 = valueAssignment;
```

Description of Syntax: the basic declaration of a new decision variable (excluding the declaration of an optional default value) consists of the desired type (one of the basic types, an enumeration, a container type, a derived or a restricted type, or a compound type) followed by an identifier (*name*$_1$) that states the name of the variable.

Optionally, a default value can be assigned to a decision variable appending "=" followed by a "*valueAssignment*"-expression after the name (*name*$_2$) of the decision variable. The form of the "*valueAssignment*"-expression depends on the specific type of the declared decision variable:

- Basic types and Enumerations: an expression that yields a value of the corresponding type and can be actually calculated, i.e., it either consists of constants or the values of the variables are known.
- Container types: either an expression of the type of the container, which can be statically evaluated, or a set of values separated by commas in curly brackets after the name of the decision variable. Expressions may be used but must be stated in parenthesis due to technical reasons. The allowed values within the curly brackets are determined based on the base type of the container.
- Compounds: either an expression of the type of the compound, which can be statically evaluated, or a set of individual assignments, given in curly brackets. Each assignment explicitly gives the field in the compound that the assignment is made to, followed by a "=" and an expression of the corresponding element type. Again this expression needs to be statically evaluated.
- Derived type: the assignment follows the rules of the base type.

Example:

```
/* Declaration of a new variable of type integer with a
default value. */

Integer bitrate = 128;


/* Declaration of a new variable of type enumeration with
a default value (cf. Section 2.1.2.2). */

Colors backgroundColor = Colors.black;
```

```
/* Declaration of a new variable of type container
(sequence) with default values (cf. Section 2.1.2.3). */

sequenceOf(ContentType) baseContent =
   {ContentType.text, ContentType.audio};



/* Declaration of a new variable of type compound with
default values (cf. Section 2.1.2.5). */

Content complexContent = {name = "Text",
  bitrate = 128};
```

2.1.4 Constraints

Constraints are used to define validity rules for a variability model, e.g. by specifying dependencies among decision variables. The syntax of constraints in the IVML basically follows the structure of expressions in propositional logic and, thus, is composed of:

- Simple sentences, which represent constants, decision variables and types which can be named by (qualified) identifiers.
- Compound sentences created by applying the operations to simple sentences and, in turn, to compound sentences. A correct compound sentence requires that the arguments passed to operations match the arity of the operation and the types of the parameters or operations comply, respectively.

The operations available in IVML as well as the type compliance rules will be discussed in the remainder of this section.

The constraints in IVML will mostly rely on the relevant part of the syntax as well as on a large subset of the operations defined in OCL (c.f. Section 3 for a description of all operations). In IVML we use the constraint expression syntax of OCL, but omit the OCL contexts used to relate constraints to UML modelling elements. Similar to OCL, all elements defined in an IVML model will be accessible to constraints. Two examples for constraints are given below, one propositional and one first-order logic example using a quantifier:

- `(10 <= a and a <= 20) implies b == a;`
  If `a` is in the range `(10; 20)` this implies that `b` must have the same value as `a`.
- `1 <= mySet.size() and mySet.size() <= 100`
  Cardinality restriction of mySet containing arbitrary decision variables.
- `mySet->forAll(x|x > 100);`
  All elements in `mySet` must be larger than `100`

Constraints may be used in two distinct ways in IVML:

- Standalone constraints: Constraints are given as statements in a project or within a compound so that compound fields are directly accessible without

qualification. As standalone constraints are used like statements, they end with a semicolon (as shown in the two examples above).
- Embedded constraints: One or more constraints are used as part of a statement, for example a **`typedef`**. Here the constraint is written in parenthesis and not ended by a semicolon.

Below we will discuss individual elements of constraints in IVML and, in particular, the difference (in particular regarding an adapted notation) to the related elements in OCL. Large parts of the remainder of this section are directly taken over from the OCL specification [4] and adapted to the IVML context.

Keywords

Keywords in IVML constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords is shown below:

- **`and`**
- **`def`**
- **`else`**
- **`endif`**
- **`if`**
- **`iff`**
- **`implies`**
- **`in`**
- **`let`**
- **`not`**
- **`or`**
- **`then`**
- **`xor`**
- **`null`**

Prefix operators

IVML defines two prefix operators, the unary

- Boolean negation '**`not`**'.
- Numerical negation '−' which changes the sign of a Real or an Integer.

Infix operators

Similar to OCL, in IVML the use of infix operators is allowed. The operators '+,' '−,' '*.' '/,' '<,' '>,' '<>' '<=' '>=', '=', '==', '!=' and '<>' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the "+" operation on *a* (the *operand*) with *b* as the parameter to the operation. The infix operators defined for a type must have exactly one parameter.

For the infix operators ‘<,’ ‘>,’ ‘<=,’ ‘>=,’ ‘<>,’ ‘**and**,’ ‘**or**,’ ‘**xor**’, ‘**implies**’, ‘**iff**’ the return type is Boolean.

Please note that, while using infix operators, in IVML Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real.

Equality and assignment operators (default logic)

In contrast to OCL, IVML provides two operators which are related to the equality of elements with different semantics, namely the default assignment ‘=’ and the equality constraint operator ‘==’. We will explain the difference in this section.

Basically, a decision variable in IVML is considered as **undefined**, i.e., the variable does not have an effect on the instantiation. Constraints may explicitly refer to the undefined state via the operation “isDefined”. Please note that for instantiation all (relevant) decision variables must be frozen (cf. Section 2.2.4.2) and that also undefined decision variables can be frozen.

A **default value** may be assigned to a variable. Default values can be used to define a basic configuration (a kind of basic profile) which applies to all products in the product line. A default value can be defined as part of the variable declaration[2] (using the ‘=’, cf. Section 2.1.3) or in terms of an individual default assignment using the ‘=’ operator. Default values may be changed by partial configuration (cf. Section 2.2.4.1), i.e. on the import path of a (hierarchical) variability model the default value of certain decision variables may be modified in order to adjust the basic profile, e.g., to a certain application setting or domain. However, a default value may only be modified (assigned or changed) once in a given model. This restriction is required due to the fact that IVML does not provide support to define the sequence of evaluations (except for imports and eval blocks, cf. Section 2.2.4.3).

As the ‘=’ operator defines a default value which may be overridden, it is not possible to use that operator to express that a decision variable must have a certain value (under some conditions). This can be achieved using the equality operator ‘==’. Basically, the equality operator checks whether the left hand and the right hand operand have **equal values**. In two distinct cases, the equality operator **enforces the value** specified by the right hand operand. The cases are the

- Unconditional value constraint, e.g., `a == 5`.
- Conditional value constraint given as the right side of an implication, e.g., `c < 5 implies a == 5`.

In these two cases, the equality operator expresses that the left hand operand (an expression denoting a decision variable) must have the same value as the right hand operand. If the left hand operand contains a default value, then the default value will be overridden. However, if two expressions aim at enforcing different values for the same decision variable, the model becomes unsatisfiable.

[2] A decision variable declaration which defines a default value is semantically equivalent to a decision variable declaration without default value and a subsequent default assignment (somewhere) in the same model.

Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot and arrow operations: '`.`' (for element and operation access) and '`->`' (to access collection operations such as **`forAll`** or **`exists`**).
- unary '**`not`**' and unary minus '`-`'
- '`*`' and '`/`'
- '`+`' and binary '`-`'
- '**`if-then-else-endif`**'
- '`<`', '`>`', '`<=`', '`>=`'
- '`==`' (equality), '`<>`', '`!=`' (alias for '`<>`')
- '**`and`**', '**`or`**' and '**`xor`**'
- Default assignment '='
- '**`implies`**', '**`iff`**'

Parentheses '`(`' and '`)`' can be used to change the precedence of operators in expressions.

Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- AnyType is the common superclass of all types. All types comply with AnyType. However, AnyType is typically used for defining the built-in operations. The only value of AnyType is **`null`**, which explicitly makes a decision variable undefined.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
- The **`refines`** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.

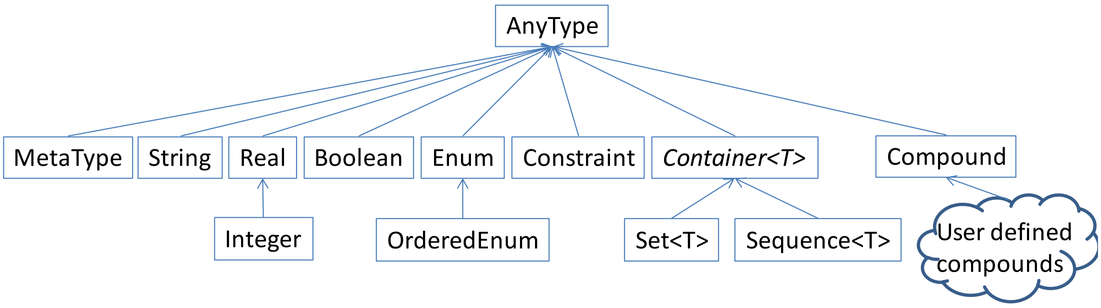

Figure 1: IVML type hierarchy

- Derived types are compliant to their base type as long as if no constraints were specified.
- MetaType is a specific type denoting types, e.g. to constrain types of elements within a collection.

Type operations

IVML provides the **isTypeOf**(), **isKindOf()** and **typeOf()** operations. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. Currently, IVML neither supports re-typing or casting.

Enumeration Types

Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with default assignment ('='), equality ('==') or unequality ('!=', '<>') operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators ('<', '>', '<=', '>=') may be used.

Compound Types

Decision variable declarations defined within a compound can be accessed using the dot operator '.'.[3]

String Type

In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes. For details please refer to Section 3.

[3] Please note that the current implementation of IVML accepts qualified and unqualified variable names within a compound while unqualified shall be default for denoting variables within the same compound. However, the current reasoning mechanism may not properly distinguish both cases so that a qualification with the compound name for variables denoting variables within a compound are required.

Configuration Type

A decision variable of type `Configuration` represents a variable constraint. Such a variable needs to be used somewhere in an IVML model in order to become active. Further statements or constraints may override the constraint in such a variable.

If-then-else-endif Expressions

The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expressions must be used within the `then` and `else` parts, both yielding the same type. The `else` part is not optional.

```
if contents[0].type == "video"
  then contents[0].bitrate
  else contents[0].highBitrate;
```

Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The **`let`** expression allows one to define a variable that can be used in the constraint. We adjusted the notation to the IVML convention so that the type is stated before the name.

```
let Integer sumBitrate = bitrates->sum()
  in sumBitrate <= 256;
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.

User-defined operations

To enable the named reuse of (larger) constraint expressions, user-defined operations can be defined. The syntax of the operation definitions is similar to the Let expression, but each attribute and operation definition is prefixed with the keyword **`def`** as shown below. We adjusted the notation as IVML does not have OCL contexts (no colon after **`def`**) and that the type is stated before the name of the operation or parameter.

```
def Integer actualBitrate(Contents c) =
   if c.type == "video"
     then c.bitrate
     else c.highBitrate;
```

The name of an operation may not conflict with keywords, types, decision variables, etc. An user-defined operation may be used similar to build-in operations. Please note that prefix or infix use of user-defined operations is not supported.

```
actualBitrate(c) > 1024 implies highQuality == true;
```

Collection operations

IVML defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of collections or projecting new collections from existing ones. However, we support only a relevant subset of the various notations in OCL. The different constructs are described in the following paragraphs. All collection operations (and only those) are accessed using the arrow-operator '->'.

In the first versions of OCL, all collection operations returned flattened collections, i.e. the entries of nested collections instead of the collections were taken over into the results. However, this was considered as an issue in OCL and does not fit to the explicit hierarchical nesting in IVML. Thus, collection operations in IVML do not apply flattening.

Sometimes an expression using operations results in a collection, while we are interested only in a special subset of the collection. The **select** operation specifies a subset of a collection:

```
collection->select(t|boolean-expression-with-t)

collection->select(ElementType t|

  boolean-expression-with-t)
```

Both expressions result in a collection that contains all the elements from `collection` for which the `boolean-expression-with-t` evaluates to true. Thereby, `t` is an iterator which will successively receive all values stored in `collection`. In the second form the type of the elements is explicitly specified. Note that the type of the iterator must comply with the element type of the collection. To find the result of this expression, for each element in `collection` the expression `boolean-expression-with-t` is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not.

Example:

```
/* Get all elements of the set "contents" with a
"highBitrate" of less than 128 */

contents->select(t|t.highBitrate < 128);
```

The **reject** operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax.

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a **collect** operation. The **collect** operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect(t|boolean-expression-with-t)
```

```
collection->collect(ElementType t|
  boolean-expression-with-t)
```

Many times a constraint is needed on all elements of a collection. The **`forAll`** operation in IVML allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll(t|boolean-expression-with-t)

collection->forAll(ElementType t|
  boolean-expression-with-t)
```

Example:

```
/* None of the elements of the set "contents" must have a
"highBitrate" of greater than 512 */

contents->forAll(t|t.highBitrate <= 512);
```

The **`forAll`** operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a **`forAll`** on the Cartesian product of the collection with itself.

```
collection->forAll(t1, t2|
  boolean-expression-with-t1-and-t2)

collection->forAll(ElementType t1, t2|
  boolean-expression-with-t1-and-t2)
```

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The **`exists`** operation in IVML allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists(t|boolean-expression-with-t)

collection->exists(ElementType t|
  boolean-expression-with-t)
```

Depending on the type of the collection further related operation may be defined such as **`isUnique`**. Details will be given in Section 3 where we describe all operations in detail.

One special case of collection operation is to aggregate one value over all values in a collection by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the **`apply`** operation while reusing the already known syntax:

```
collection->apply(t, ResultType r = initial|
  r = expression-with-t)
```

```
collection->apply(ElementType t, ResultType r = initial|
  r = expression-with-t)
```

This operation initializes the result "iterator" `r` with the `initial` expression and applies the `expression-with-t` to each element in the collection. The result of `expression-with-t` is used to update successively the result "iterator". Finally, the operation returns the value of `r` after processing the last element in `collection`. Please note that the result "iterator" is always defined using a specific type which, in turn, defines the result type of the apply operation.

Example:

```
/* Return the sum of all (default) bitrates of the
elements of the set "contents" */

contents->apply(t, Integer r| r == t.bitrate);
```

2.1.5 Configurations

The IVML does not differentiate between a configuration space and specific (product) configurations. Instead, a project can simultaneously describe or extend a configuration space and define a configuration. However, typically a project will provide a configuration space, while a different project may extend it, while providing configurations information for the initially specified configuration space. The set of decision variables and constraints of a project represent the set of all possible configurations. In addition, default values of decision variables as described in Section 2.1.3 define basic configurations and, thus, do not need to be further configured, but can be overwritten later as well. In addition, some values of decision variables can be derived using constraints. Any configuration, independent of where the values come from, must comply with the relevant constraints.

Configurations in the IVML do not require any specific or additional keyword. They are simply given by variable assignments. We illustrate this concept by a simple example.

Example:

```
/* A project that represents both a configuration space
and a configuration. The constraint implies a valid
configuration with a bitrate value between "128" and "256"
and "content == text" (if no further configuration is
done). */

project contentSharing {

  enum ContentType {text, video, audio, threeD, blob};

  typedef Bitrate Integer with (Bitrate >= 128 and
    Bitrate <= 256);

  ContentType content;
```

```
    Bitrate contentBitrate = 128;

    contentBitrate == 128 implies
      content == ContentType.text;

}
```

2.2 Advanced Concepts of the INDENICA Variability Modelling Language

This section describes advanced concepts of the IVML. We will describe how to assign additional attributes to modelling elements. This allows describing certain modelling elements in more detail, e.g. assigning meta-variability information as described in D2.1. We then augment the compound types introduced in Section 2.1.2.5 by extension and referencing concepts. Extension concepts will also be introduced for projects (cf. Section 2.1.1), which cover modularization aspects as well as facilitating project composition. We will describe advanced configuration concepts including partial configurations as well as "freezing" configurations. Finally, we will describe a lightweight concept for including DSLs as part of a variability model.

2.2.1 Attributes

In the IVML modelling elements can be attributed by further (orthogonal) configuration capabilities, e.g. to express meta-variability such as binding times. An attribute in IVML is basically a decision variable that is attached to another modelling element describing this element in more detail. Thus, an attribute may also have a default value and may be restricted by constraints (cf. Section 2.1.4). The impact of an attribute depends on the element it is attached to. In the IVML the following modelling elements can be attributed:

- Decision variable: attributes that are attached to a decision variable only describe this variable further. Depending on the type of the decision variable, the attributes of the variable also describe its elements, e.g. the various fields of a compound variable. These fields may have additional attributes. Changing the value of a decision variable attribute will not cause any modification to elements outside the scope of the specific variable (as far as they are not connected by constraints).
- Project: attributes that are attached to a project will affect all variables of this project.

As the different elements may be nested, different values can be given for the same attribute on the outer and the inner scope.

Syntax:

attribute *Type* $name_1$ **to** $name_2$;

attribute *Type* $name_3$ = *value* **to** $name_4$;

Description of Syntax: the definition of an attribute consists of the following elements:

- The **`attribute`** keyword indicates the definition of a new attribute.
- The expressions *`Type name`*$_1$ and *`Type name`*$_3$ correspond to the definition of a decision variable described in Section 2.1.3 while *`name`*$_1$ and *`name`*$_3$ are the identifiers of the new attributes[4].
- The **`to`** keyword indicates the attachment of the new attribute on the left side to the element (*`name`*$_4$) denoted on the right side. Multiple names may be given separated by commas
- *`name`*$_4$ may be one of the elements described above to which the attribute is attached.
- Optionally, a default value (*`value`*) can be assigned to the attribute by appending a value expression after *`name`*$_3$.

Example:

```
project contentSharing {

  enum BindingTimes {configuration=0, compile=1,
     runtime=2};

  // Attaching an attribute to the entire project.

  attribute BindingTimes binding = BindingTimes.compile
    to contentSharing;

}
```

Attributes can also be used in initializing expressions for containers and compounds. This is demonstrated in the fragment below:

```
compound Content {

  String name;

  Integer bitrate;

}

Content content;

attribute BindingTimes binding = BindingTimes.compile
    to content;
```

[4] Due to technical reasons, currently attributes must not start with ‘v’ or ‘e’.

```
content = {name="Video", bitrate=128,
     name.binding=BindingTimes.compile,
     bitrate.binding=BindingTimes.runtime};
```

However, assigning the same value for a certain attribute for a given set of decision variables may increase the perceived complexity of the model as similar assignments are repeated.

Example:

```
project contentSharing {

  enum BindingTimes {configuration=0, compile=1,
     runtime=2};

  // Attaching an attribute to the entire project.

  attribute BindingTimes binding = BindingTimes.compile
    to contentSharing;

  enum Colors {black, white};

  Bitrate contentBitrate = 128;

  contentBitrate.binding = BindingTimes.configuration;

  Colors backgroundColor = Colors.black;

  backgroundColor.binding = BindingTimes.configuration;

  // go on with several variables and different binding

  // times

}
```

IVML provides the assign construct as syntactic sugar to simplify the mass-assignment of values to attributes and to visually group the model elements with same (initial) attribute assignment. However, the variables "declared" in the assign block actually are part of the containing element, in the example below the project `contentSharing`. An assign block can also be used within compounds, it may even be nested in other assign blocks if needed or multiple attributes may be given in comma-separated fashion in the parenthesis of an assign block. As an assign block is technically translated into individual assignment constraints ('=') as stated as a generic constraint in the parenthesis of an assign block.

Example:

```
project contentSharing {
```

```
    enum BindingTimes {configuration=0, compile=1,
       runtime=2};

    // Attaching an attribute to the entire project.

    attribute BindingTimes binding = BindingTimes.compile
      to contentSharing;

    enum Colors {black, white};

    assign (binding = BindingTimes.configuration) to {

      Bitrate contentBitrate = 128;

      Colors backgroundColor = Colors.black;

      // go on with the variables of the same binding time

    }

}
```

2.2.2 Advanced Compound Modelling

In Section 2.1.2.5 we introduced the compound types to group multiple types into a single named unit. In this section, we will extend the modelling of compound types by refinement and referencing concepts. Refinement allows extending existing compound types by additional elements, yielding a new (extended) compound type. Referencing enables the definition of references to other elements like other compounds.

2.2.2.1 Extending Compounds

In the IVML a compound may extend the definition of a previously defined (parent) compound. This is indicated by the **`refines`** keyword. Extending compound types is similar to subclassing in object-oriented languages, i.e. *`parentType`* becomes a subtype of *`compoundType`* and *`compoundType`* may define further decision variables.

Syntax:

```
compound Name1 refines Name2 {

  // Define additional elements.

}
```

Description of Syntax: the definition of an extended compound type consists of the following elements:

- The **`compound`** keyword indicates the definition of a new compound type.
- The identifier *`Name`*$_1$ defines the name of the new compound type.
- The **`refines`** keyword indicates that the new compound type (*`Name`*$_1$) is an extension of a previously defined compound type (*`Name`*$_2$).

- The set of elements surrounded by curly brackets defines the additional elements that make up the extensions to the inherited elements of compound *Name*$_2$.

Example:

```
/* A compound type for the configuration of different (web) content. */

compound Content {

  String name;

  Integer bitrate;

}


/* A new compound type that refines the previous compound type. "ExternalContent" will subsume all elements of "Content" and all additional elements defined below. */
compound ExternalContent refines Content {
  String contentPath;
  String accessPassword;
}
```

2.2.2.2 Referencing Elements

The IVML supports referencing of (other) elements, for example, other compounds within a compound type. A reference allows the definition of individual configurations of an (external) element for the referencing element without including the external element as part of the referencing element explicitly. This is indicated by the **`refTo`** keyword used for the definition of a reference and the **`refBy`** keyword that indicates the configuration of a referenced element.

Syntax:

```
project name1 {

  compound Name2 {

    Type name3;

    ...

  }


  // Declaration of a new reference.

  refTo(Name2) Name4;
```

```
    // Configuration of a referenced element.
    refBy(Name4).name3 = value;
}
```

Description of Syntax: the definition and the configuration of a reference consist of the following elements:

- The **refTo** keyword indicates the definition of a new reference.
- $Name_2$ defines the referenced element (type).
- $Name_4$ is an identifier and defines the name of the new reference. In the IVML a reference is type, thus, the identifier for a new reference starts with a capital letter.
- The **refBy** keyword indicates the configuration of a reference (the configuration of the referenced element respectively).
- $Name_4$ is an identifier that defines the reference to be configured.
- The syntax for configuring a reference depends on the type of the referenced element (see Section 2.1.3 for the syntax for assigning values to variables of a specific type). In the case above, we use "`.`"-notation to configure a single element of a referenced compound type.

Example:

```
/* A compound type for the configuration of different web
containers being responsible for serving web content. */
compound Container {
    String name;
    ...
}


/* Another compound type for the configuration of
different (web) content referencing the "Container" type
to configure its individual web container. */
compound Content {
    String name;
    Integer bitrate;
```

```
    // Declaration of a reference to the Container compound.
    refTo(Container) myContainer;


    // Configuration of the above reference.
    refBy(myContainer).name = "ContentContainer";
}
```

2.2.3 Advanced Project Modelling

In Section 2.1.1, we introduced the concept of projects (**`project`**) as the top-level element in each IVML-model. In this section, we extend the modelling capabilities of the IVML regarding projects in three ways: first, we describe versioning of projects that enables the definition of the current state of evolution of a project. This concepts correlates with the second concept: project composition. This introduces the capability of deriving new projects based on definitions in other projects and explicitly excluding certain projects from the composition. As part of this version information can be used. The third concept is project interface. The concepts of project composition and project interfaces support effective modularization and reuse of projects and, thus, configuration spaces.

2.2.3.1 Project Versioning

In IVML, projects can be versioned to define the current state of evolution of a project (and the represented product line infrastructure). Evolution of software may yield updates to projects. This can be described by a version. For defining a version, the **`version`** keyword is followed by a version number. This must be the very first element of the respective project. The version number consists of integer values separated by "`.`" assuming that the first value defines the major version, while following numbers indicate minor versions. The level of detail of version numbers is determined by the domain engineer.

Syntax:

```
project name {
    // Definition of a version for this project
    version vNumber.Number;
    ...
}
```

Description of Syntax: the attachment of a version to a project consists of the following elements:

- The **`version`** keyword indicates the definition of a new version for the project *`name`*.

- *vNumber.Number* defines the actual version of the project (here only two parts prefixed by a "v"). At least one number must be given and no restriction holds on the amount of sub-version numbers.

Example:

```
project contentSharing {
  version v1.0;
  ...
}
```

2.2.3.2 Project Composition

The IVML supports the composition of different projects. This is closely related to multi software product lines [8] and product populations [9]. Project composition allows to effectively reusing existing projects by using these projects within other projects. This also supports the decomposition of large variability models as semantically related parts can be defined in individual projects. The complete project then uses these (sub-) projects to define the combined project. In the IVML the following keywords are introduced for project composition:

- **`import`**: this keyword indicates the use of a project. An imported project is evaluated before import, thus an import acts as an implicit eval. This keyword allows using certain elements of a project by reference. If a project contains explicit interfaces (see below), the specific interface, which is used, must be given.
  However, multiple projects with identical names and versions may exist in a file system[5], in particular in hierarchical product lines. Thus, project imports are determined according to the following **hierarchical import convention**, i.e. starting at the (file) location of the importing project (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). In addition, sibling folders of the folder containing the importing model and predecessor projects are considered[6]. Similar to Java class paths, additional model paths[7] may be considered in addition to the immediate file hierarchy.
- **`conflicts`**: this keyword indicates incompatibility among projects. All projects (names) followed by this keyword cannot be used in combination with the project that defines this conflict expression. This is also checked for

[5] The implementation of the tool support decides whether the entire file system or a subtree is considered. In EasY-Producer, currently the entire active workspace is considered.

[6] Actually, EasY-Producer stores the imported parent product line models in individual subfolders (starting with a "."), i.e. possibly sibling folders of a model.

[7] The actual implementation is already prepared for model paths. Depending on the actual use we will include model paths into the user-level of the tool support.

indirectly used projects. Also project names in conflicts are resolved according to the hierarchical import convention defined above.

The keywords **`import`** and **`conflicts`**, introduced above, can be combined with version expressions using the **`with`** keyword and the version-information of a project introduced in Section 2.2.3.1. Note, that versions restrictions are no fully-fledged constraints and only the relational operators ‘<’, ‘>’, ‘<=’, ‘>=’ as well as the equality operators ‘=’, ‘==’, ‘<>’, ‘!=’ may be used here. Please note that version numbers start with “v” (cf. Section 2.2.3.1).

Syntax:

```
project name1 {

  /* This introduces the project name2. Optionally, a
  version may restrict name2 to a specific version as it
  is shown below. */

  import name2;

  // Accessing elements of a project.

  name2::element;


  /* This introduces incompatibility of project name1 with
  project name3 of version greater than Number.Number. */

  conflicts name3 with (name3.version > vNumber.Number);

}
```

Description of syntax: the definition of a new project composition consists of the following elements:

- The keyword **`import`** indicates that the entities, which are made available by the project or interface *name*$_2$ will be available within the current project.
- For disambiguation the elements of *name*$_2$ can be accessed using the “`::`”-notation to express qualified names. If there is no ambiguity, they can be used directly.
- The keyword **`conflicts`** indicates incompatibility of project *name*$_1$ with project *name*$_3$.
- Optionally, version-expressions can be combined with the keywords **`import`** and **`conflicts`** using the **`with`** keyword. This defines specific versions of other projects to be imported into the current project or conflicting with the current project.

- A version expression includes the version-information of a project (cf. Section 2.2.3.1), a relation operator and a version number or a version-information of another project. In addition, logical operators can be used to concatenate simple version-expressions to define ranges of versions.

Example:

```
project application {

  /* This will define a new project for content-sharing
  applications. */

  String name;

}


project targetPlatform {

  // This will define a new project for target platforms.

  version v1.5;

  String name;

}


project contentSharing {

  /* This will define a new project for a content-sharing
  project importing two sub-projects "application" and
  "targetPlatform". The latter sub-project must be of
  version "1.3" or higher. */

  import application;

  import targetPlatform
    with (targetPlatform.version >= v1.3);


  // Accessing the elements of the sub-projects.

  application::name = "myApp";

  targetPlatform::name = "myPlatform";

}
```

2.2.3.3 Project Interfaces

By default, all elements defined in a project are visible when they are imported into another project. In order to support effective modularization and reuse of variability models, we introduce interfaces to projects. Interfaces reduce the complexity in large-scale projects and provide means to automate the configuration of lower-level decisions based on high-level decisions.

Interfaces in a project define all elements of a project, not part of the interface, as private and, thus, make them invisible to the outside. This is indicated by the **`interface`** keyword within a project. In order to access any elements they need to be declared as parameters of the interface. This can be done by exporting existing variables (using the **`export`** keyword) or by declaring new parameter variables. As a special characteristic of the IVML, it is also possible to define multiple interfaces for the same project. This is different from other variability modelling languages like the CVL [6].

Importing a project (cf. Section 2.2.3.2) that includes interfaces allows the importing project to access only the parameters defined in the interface. All other elements of the project are not visible to the importing project.

Syntax:

```
project name1 {
  // Definition of a new interface.
  interface Name2 {
    /* Denotes the export of an existing decision variable
    of the project name1. */
    export name3;
    ...
  }
  /* Declaration of a (private) decision variable. This
  variable is exported by the interface Name2. */
  Type name3;
}
```

Description of syntax: the definition of a new project interface consists of the following elements:

- The keyword **`interface`** indicates the definition of a new interface of the project *$name_1$*. Interfaces must occur at the beginning of a project before decision variable or type definitions.

- The keyword **`export`** indicates the export of the following decision variable *name*$_3$.

Example:

```
project application {
  // This will define an interface for this project.
  interface MyInterface {
      export name, appType;
  }

  // Declaration of (private) decision variables.
  String name;
  String appType;
  Integer bitrate;

  // Definition of a constraint.
  appType == "Video" implies bitrate == 256;
}

project contentSharing {
  /* This will import the interface "MyInterface" of
  project "application". */
  import application::MyInterface;

  /* Only the parameters of the interfaces are accessible.
  "application::bitrate" yields an error. As long as the
  variable names are unambiguous, the fully qualified must
  not be used. */
  name = "myApp";
  appType = "Video";
}
```

2.2.4 Advanced Configuration

In Section 2.1.5, we introduced the configuration concept of the IVML. In this section, we will extend this concept to partial configuration. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yield a full configuration of the entire project. We will further introduce the concept of persistent (parts of) configurations. We call this “freezing”. Freezing (parts of) configurations defines these parts to be persistent. Persistent parts cannot be changed anymore in further configuration steps. Finally, we will describe how (parts of) configurations can be evaluated independently from other parts of the configuration. This allows deriving additional configuration values based on existing configurations using the constraints and value propagation.

2.2.4.1 Partial Configurations

The IVML supports partial configurations. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yields a full configuration of the entire project. The configuration of a part of a project may also be reconfigured by the next configuration step (cf. the concept of default values, which we introduced in Section 2.1.3). For example, a service provider may define a (pre-) configuration of the provided service, while a service consumer may reconfigure his service to satisfy his specific needs.

Partial configuration in the IVML is a straight-forward consequence of the concepts introduced so far. We illustrate this concept by a simple example.

Example:

```
project application {

  /* This defines a new project for content-sharing
  applications including the (pre-) configuration of the
  configuration element. This is also the first
  configuration step.*/

  String name = "Application";

}


project targetPlatform {

  /* This defines a new project for target platforms
  without any configuration. */

  String name;

}
```

```
project contentSharing {

  /* This defines a new project for a content-sharing
  project and imports two sub-projects "application" and
  "targetPlatform". */

  import application;

  import targetPlatform;


  /* This is the second configuration step, including the
  re-configuration of the name-element of the sub-project
  "application" and a configuration of the name-element of
  the sub-project "targetPlatform". */

  application::name = "myApp";

  targetPlatform::name = "myPlatform";

}
```

2.2.4.2 Freezing Configurations

In the previous section we described the concept of partial configuration. This included the possibility to re-configure existing (pre-) configurations. Although re-configuration is reasonable in some cases, e.g. to modify a given configuration to satisfy an individual need, at the end we desire a persistent configuration to define a specific product. For example, service consumers should not be able to reconfigure some parts of a configuration defined by a service provider.

We introduce the concept of "freezing" configurations. This is indicated by the keyword **`freeze`**. Freezing configurations define the current (partial) configuration to be persistent. Persistent configurations cannot be changed anymore in the course of the configuration. Excluding elements of a configuration from being frozen, e.g. freezing only some elements of imported projects or a compound type, the **`but`** keyword can be attached after a freeze-expression. All elements followed by a but-expression will not be frozen.

Freezing an undefined variable *v* leaves *v* undefined so that *v* does not have an effect. In particular, *v* may be changed afterwards and *v* may be part of a configuration implicitly disabling some instantiation.

Syntax:

```
project name1 {

  // Definition of new compound type

  compound Name2 {

    Type name3;
```

```
    Type name4;
  }

  /* Declaration of a new decision variable of the above
  type */
  Name2 name6;

  /* Freezing the configuration of the decision variable
  except element name4. */
  name6.name3 = value1;
  freeze {
    name6;
  } but (name6.name4)
}
```

Description of syntax: the definition of persistent (parts of) configurations consists of the following elements:

- The keyword **`freeze`** indicates that all elements with their current values within the following curly brackets are persistent.
- Optionally, the keyword **`but`** indicates a set of elements that is excluded from being persistent. All elements of this set can be further configured. The but-expression may also include wildcards (*) which are necessary especially in large models. Attaching a wildcard to an element, e.g. $name_6$`.*`, yields all elements of $name_6$ to be excluded from being frozen.

Example:

```
project application {
  /* Definition of a new compound type for the
  configuration of the content type of an application. */
  compound ContentType {
    String contentName;
    Integer bitrate;
  }
  // Declaration of a decision variable of the above type.
```

```
  ContentType appContent;

  /* Definition of the content name to be persistent. The
  required bitrate for this content may be configured as
  part of the configuration of the container type for this
  content. */

  appContent.contentName = "Text";

  freeze {

    appContent;

  } but (appContent.bitrate)

}
```

2.2.4.3 Partial Evaluation

The IVML provides a concept for the evaluation of configurations. This is indicated by the keyword **`eval`**. The explicit declaration of *nested* **`eval`** structures can be used to structure the definition of the variables and thus reduces the search-space during constraint-evaluation. By default, the top-level **`eval`** structure is the containing project, i.e., at the end of a project definition an implicit **`eval`** occurs as the project is the topmost eval. **`eval`** structures on the same nesting level do not imply a sequence of evaluation as this is true for the constraints in a project.

Currently, an eval statement may only contain constraints, i.e., variables are project global and no variables can be defined in an eval (this may change in future, then variables would be propagated from inside the eval the outside eval or project).

Syntax:

```
/* Evaluate a constraint that defines the relation between
two variables of the same type. This leads to the
assignment of the variable values to the unassigned
variable upon exit of the scope of the eval-statement.
Note that this eval is evaluated before any other
constraint in the project is evaluated.*/



eval {

  name1 = name2;

}
```

Description of syntax: the evaluation of a configuration requires an **`eval`**-statement using the keyword **`eval`** followed by curly brackets.

Example:

```
project application {

  /* Definition of a new compound type for the
  configuration of the content type of an application. */

  compound ContentType {

    String contentName;

    Integer bitrate;

  }

  // Declaration of a decision variable of the above type.

  ContentType appContent;

  /* Definition of the content name and bitrate. This
  configuration is evaluated explicitly to minimize the
  search space. */

  eval {

    appContent.contentName == "Text" implies
      appContent.bitrate = 128;
  }

}


project targetPlatform {

  /* Define a new project for target platforms without any
  configuration.*/

  String name;

  Integer bitrate;

}


project contentSharing {

  /* Define a new project for a content-sharing project
  importing two sub-projects "application" and
  "targetPlatform".*/

  import application;

  import targetPlatform;
```

```
/* This constraint restricts the bitrate of the target
platform to be equal or greater than the bitrate of the
application content. The bitrate of the target platform
can be derived from the bitrate of the application
content: "targetPlatform::bitrate == 128". At the end of
a project definition an implicit evaluation for the
whole project is done. */

targetPlatform::bitrate
  >= application::appContent.bitrate;

}
```

2.2.5 Including DSLs

The IVML includes a lightweight concept for including domain-specific languages (DSLs) as part of the variability model. This supports situations, in which the variability may be expressed more intuitively or more naturally using DSLs.

DSLs can be embedded in IVML in terms of external language sections similar to inline assembler code in higher languages. The embedded DSL code is preprocessed in order to consider actual decision values during DSL evaluation, passed to a DSL-specific tool for evaluation and the result of the evaluation is considered as part of the actual IVML model, which triggered the evaluation. The evaluation result is interpreted as a part of the final IVML description.

Syntax[8]**:**

```
DSL(stopString, prefix, dslInterpreter) %

  // here goes the DSL

DSL%;
```

Description of syntax: an external language section for a DSL is introduced by the keyword **`DSL`** and closed by **`DSL%`**. The parameters of the opening **`DSL`** keyword are:

- The *`stopString`* identifier[9] is a string used for uniquely identifying the end of the DSL in combination with the **`DSL`** keyword. The part between the opening **`DSL`** keyword (excluding its parameters in parentheses) and the closing **`DSL`** keyword (marked by the *`stopString`*) is not analyzed by the IVML tools but passed to an external DSL interpreter for evaluation.
- The *`prefix`* identifier is a string identifying a DSL-specific prefix for IVML identifiers denoting decision variables. When passing the DSL code to the

[8] Technically, a DSL fragment is implemented as an expression of AnyType. AnyType is the common supertype in the type system of OCL and IVML. However, in IVML AnyType cannot be used in subexpressions, i.e. a DSL fragment is written as a standalone expression statement while the use within an expression is syntactically but not semantically correct.

[9] Due to technical reasons implementing this concept in xText, the *`stopString`* is now fixed to **`DSL%`**. The parameter *`stopString`* is kept as parameter for legacy reasons but may be subject to removal in future versions.

DSL specific tools, all occurrences of decision variables marked by the *prefix* are replaced by actual values for the individual decisions.

- The *dslInterpreter* identifier is a string containing, for example, a file name or an URI specifying the concrete DSL tool which is responsible for evaluating the instantiated DSL code, i.e. after substituting occurrences of decision variables.

Example:

```
project application {

  /* Declaration of a decision variable with a default
  value. */

  Integer bitrate = 128;


  /* Declaration of an embedded DSL section within an IVML
  project. */

  DSL("dsl.com","$","http://www.dsl.com/dslInterpreter") %

    /* The actual DSL statements will be placed between
    the DSL keywords. */

    ...

    /* Applying IVML decision variables to DSL statements
    by using the DSL-specific prefix "$" defined above. */

    ... $bitrate ...

  DSL%;

}
```

3 Constraints in IVML

In this section we will describe syntax and semantics of the IVML constraint sublanguage. In Section 3.1 we will describe the constraint language and in Section 3.2 the built-in operation which can be used within constraint expressions.

3.1 IVML constraint language

In this section we will define the syntax and the semantics of the IVML constraint language. As constraints in IVML heavily rely on OCL, most of the content in this section is taken from OCL [4] and adjusted to the notational conventions and the semantics of IVML.

3.1.1 Keywords

Keywords in IVML constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords is shown below:

- **`and`**
- **`def`**
- **`else`**
- **`endif`**
- **`if`**
- **`iff`**
- **`implies`**
- **`in`**
- **`let`**
- **`not`**
- **`or`**
- **`then`**
- **`xor`**

3.1.2 Prefix operators

IVML defines two prefix operators, the unary

- Boolean negation '**`not`**'.
- Numerical negation '−' which changes the sign of a Real or an Integer.

3.1.3 Infix operators

Similar to OCL, in IVML the use of infix operators is allowed. The operators '+,' '−,' '`*`.' '/,' '<,' '>,' '<>' '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the "+" operation on a (the *operand*) with b as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<,' '>,' '<=,' '>=,' '<>,' '**and**,' '**or**,' '**xor**', '**implies**', '**iff**' the return type must be Boolean.

Please note that, while using infix operators, in IVML integer is a subclass of real. Thus, for each parameter of type real, you can use integer as the actual parameter. However, the return type will always be real. We will detail the operations on basic types in Section 3.4.

3.1.4 Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot and arrow operations: '.' (for element and operation access) and '->' (to access collection operations such as **forAll** or **exists**).
- unary '**not**' and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- '**if-then-else-endif**'
- '<', '>', '<=', '>='
- '==' (equality), '<>', '!=' (alias for '<>')
- '**and**', '**or**' and '**xor**'
- Default assignment '='
- '**implies**', '**iff**'

Parentheses '(' and ')' can be used to change precedence.

3.1.5 Datatypes

All datatypes defined in IVML including the user-defined ones such as compounds, restricted types or attributes are available to the constraint language and may be used in constraint expressions. Below, we give some specific notes on the use of datatypes, in particular in relation to OCL.

- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes. For details please refer to Section 3.
- Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with assignment ('='), equality ('==') or inequality ('!=', '<>') operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators ('<', '>', '<=', '>=') may be used.
- Decision variable declarations defined within a compound can be accessed using the dot operator '.'.

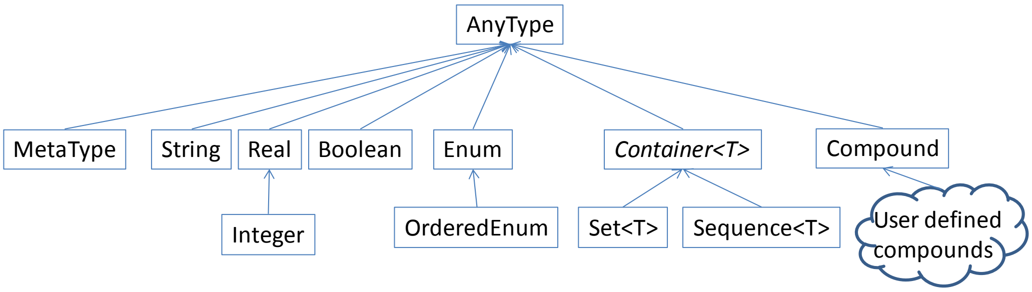

Figure 2: IVML type hierarchy

3.1.6 Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- AnyType is the common superclass of all types. All types comply with AnyType. However, AnyType is typically used for defining the built-in operations.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
- The **`refines`** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.
- Derived types are compliant to their base type as long as if no constraints were specified.
- MetaType is a specific type denoting types, e.g. to constrain types of elements within a collection.

3.1.7 Type operations

IVML provides the following type-specific operations: **isTypeOf**(), **isKindOf()** and **typeOf()**. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. In addition, the collections provide the operations **typeSelect** and **typeReject** which select elements from a collection according to their actual type based on the **isTypeOf** operation. Currently, IVML neither supports re-typing or casting.

3.1.8 Side effects

IVML is designed as a modelling and configuration language for Software Product Lines. As a configuration language, an assignment of values to decision variables is mandatory. Thus, in contrast to OCL, some constraint expressions in IVML may lead to side effects in terms of value assignments ('='). Please note that all operations except for assignments are free of side effects (similar to OCL).

3.1.9 Undefined values

Basically, variables are undefined in order to enable partial configuration. Unless a default value ('=') or a value (via the constraint operator '==') is assigned. Due to undefined variables, some expressions will, when evaluated, have an undefined value. During evaluation, undefined (sub-) expressions are ignored.

3.1.10 If-then-else-endif Expressions

The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expressions must be used within the `then` and `else` parts, both yielding the same type. The `else` part is not optional.

```
If contents[0].type == "video"

  then contents[0].bitrate

  else contents[0].highBitrate;
```

3.1.11 Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The **`let`** expression allows one to define a variable that can be used in the constraint. We adjusted the notation to the IVML convention so that the type is stated before the name.

```
let Integer sumBitrate = bitrates->sum()

  in sumBitrate <= 256;
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.

3.1.12 User-defined operations

To enable the named reuse of (larger) constraint expressions, user-defined operations can be defined. The syntax of the operation definitions is similar to the Let expression, but each attribute and operation definition is prefixed with the keyword **`def`** as shown below. We adjusted the notation as IVML does not have OCL contexts (no colon after **`def`**) and that the type is stated before the name of the operation or parameter.

```
def Integer actualBitrate(Contents c) =

   If c.type == "video"
```

```
        then c.bitrate
        else c.highBitrate;
```

The name of an operation may not conflict with keywords, types, decision variables, etc. An user-defined operation may be used similar to build-in operations. Please note that prefix or infix use of user-defined operations is not supported.

```
actualBitrate(c) > 1024 implies highQuality == true;
```

3.1.13 Collection operations

IVML defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of collections or projecting new collections from existing ones. However, we support only a relevant subset of the various notations in OCL. The different constructs are described in the following paragraphs. All collection operations (and only those) are accessed using the arrow-operator '->'.

In the first versions of OCL, all collection operations returned flattened collections, i.e. the entries of nested collections instead of the collections were taken over into the results. However, this was considered as an issue in OCL and does not fit to the explicit hierarchical nesting in IVML. Thus, collection operations in IVML do not apply flattening.

Sometimes an expression using operations results in a collection, while we are interested only in a special subset of the collection. The **select** operation specifies a subset of a collection:

```
collection->select(t|boolean-expression-with-t)

collection->select(ElementType t|
  boolean-expression-with-t)
```

Both expressions result in a collection that contains all the elements from `collection` for which the `boolean-expression-with-t` evaluates to true. Thereby, `t` is an iterator which will successively receive all values stored in `collection`. In the second form the type of the elements is explicitly specified. Note that the type of the iterator must comply with the element type of the collection. To find the result of this expression, for each element in `collection` the expression `boolean-expression-with-t` is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not.

Example:

```
/* Get all elements of the set "contents" with a
"highBitrate" of less than 128 */

contents->select(t|t.highBitrate < 128);
```

The **reject** operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax.

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a **collect** operation. The **collect** operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect(t|boolean-expression-with-t)

collection->collect(ElementType t|

  boolean-expression-with-t)
```

Many times a constraint is needed on all elements of a collection. The **forAll** operation in IVML allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll(t|boolean-expression-with-t)

collection->forAll(ElementType t|

  boolean-expression-with-t)
```

Example:

```
/* None of the elements of the set "contents" must have a
"highBitrate" of greater than 512 */

contents->forAll(t|t.highBitrate <= 512);
```

The **forAll** operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a **forAll** on the Cartesian product of the collection with itself.

```
collection->forAll(t1, t2|

  boolean-expression-with-t1-and-t2)

collection->forAll(ElementType t1, t2|

  boolean-expression-with-t1-and-t2)
```

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The **exists** operation in IVML allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists(t|boolean-expression-with-t)

collection->exists(ElementType t|

  boolean-expression-with-t)
```

Depending on the type of the collection further related operation may be defined such as **`isUnique`**. Details will be given in Section 3 where we describe all operations in detail.

One special case of collection operation is to aggregate one value over all values in a collection by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the **`apply`** operation while reusing the already known syntax:

```
collection->apply(t, ResultType r = initial|
  r = expression-with-t)
collection->apply(ElementType t, ResultType r = initial|
  r = expression-with-t)
```

This operation initializes the result "iterator" `r` with the `initial` expression and applies the `expression-with-t` to each element in the collection. The result of `expression-with-t` is used to update successively the result "iterator". Finally, the operation returns the value of `r` after processing the last element in `collection`. Please note that the result "iterator" is always defined using a specific type which, in turn, defines the result type of the apply operation.

Example:

```
/* Return the sum of all (default) bitrates of the elements of the set "contents" */
contents->apply(t, Integer r| r == t.bitrate);
```

3.2 Built-in operations

Similar to OCL, in the IVML constraint language all operations are defined on individual IVML types and can be accessed using the "." operator, such as `set.size()`. However, this is also true for the equality, relational and mathematical operators but they are typically given in alternative infix notation, i.e. 1 + 1 instead of 1.+(1). Further, the unary negation is typically stated as prefix operator. Iterative collection operations such as forAll are the only[10] operations in IVML which are accessed by "->". However, IVML also defines some specific operations which are also listed with their defining type below.

In this section, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, similar to the declaration of decision variables in IVML, we use in this section the Type-first notation to describe the signatures of the operation.

[10] This is due to technical restrictions realizing IVML with Xtext.

3.3 Internal Types

3.3.1 AnyType

AnyType is the most common type in the IVML type system. All types in IVML are subclasses of AnyType, i.e. they are type compliant and inherit the operations listed below.

- **Boolean == (AnyType a)**
  True if the *operand* is the same as *a*. This operation is interpreted as a value assertion if it is used standalone (empty implication) or on the right side of an implication. It is interpreted as an equality test if used on the left side of an implication.
- **Boolean <> (AnyType a)**
  True if the *operand* is different from *a*.
- **Boolean != (AnyType a)**
  True if the *operand* is a different object from *a*. Alias for !=.
- **MetaType typeOf ()**
  The type information of the actual type.
- **Boolean isTypeOf (MetaType type)**
  True if the *type* and the actual type of *operand* are the same. This operation can be seen as an alias for typeOf() == *type*.
- **Boolean isKindOf (MetaType type)**
  True if *type* is either the direct type or one of the supertypes of the actual type of the *operand*.

3.3.2 MetaType

MetaType represents the actual type of an object such as a specific user-defined container. Currently, MetaType inherits all operations from AnyType except for the typeOf, isTypeOf and isKindOf operations.

3.4 Basic Types

3.4.1 Real

The basic type Real represents the mathematical concept of real following the Java range restrictions for double values. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

- **Real + (Real r)**
  The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*.
- **Real * (Real r)**
  The value of the multiplication of the *operand* and *r*.
- **Real - ()**
  The negative value of the *operand*.
- **Real / (Real r)**

The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.

- **Real abs()**
  The absolute value of the *operand*.
- **Integer floor ()**
  The largest integer that is less than or equal to the *operand*.
- **Integer round()**
  The integer that is closest to *the operand*. When there are two such integers, the largest one.
- **Real max (Real r)**
  The maximum of the *operand* and *r*.
- **Real min (Real r)**
  The minimum of the *operand* and *r*.
- **Boolean < (Real r)**
  True if the *operand* is less than *r*.
- **Boolean > (Real r)**
  True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
  True if the *operand* is greater than or equal to *r*.
- **Boolean = (Real r)**
  Assigns the value *r* to the variable *operand* and returns *true*[11].

3.4.2 Integer

The standard type Integer represents the mathematical concept of integer following the Java range restrictions for integer values. Note that Integer is a subclass of Real.

- **Integer - ()**
  The negative value of the *operand*.
- **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**
  The value of the subtraction of *i* from the *operand*.
- **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*.
- **Real / (Integer i)**
  The value of the *operand* divided by *i*. Leads to an evaluation error if *i* is equal to zero.
- **Integer abs()**
  The absolute value of the *operand*.
- **Integer div (Integer i)**
  The number of times that *i* fits completely within the *operand*.

[11] The Boolean return type is required as stand-alone constraints must be of Boolean type. The result of an assignment operation is always `true`.

- **Integer mod (Integer i)**
  The result is the *operand* modulo *i*.
- **Integer max (Integer i)**
  The maximum of the *operand* and *i*.
- **Integer min (Integer i)**
  The minimum of the *operand* and *i*.
- **Boolean < (Integer i)**
  True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
  True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
  True if the *operand* is less than or equal to *i*.
- **Boolean >= (Integer i)**
  True if the *operand* is greater than or equal to *i*.
- **Boolean = (Integer i)**
  Assigns the value *i* to the operand and returns *true*[11].

3.4.3 Boolean

The basic type Boolean represents the common true/false values.

- **Boolean or (Boolean b)**
  True if either *self* or *b* is true.
- **Boolean xor (Boolean b)**
  True if either *self* or *b* is true, but not both.
- **Boolean and (Boolean b)**
  True if both *b1* and *b* are true.
- **Boolean not ()**
  True if *self* is false and vice versa.
- **Boolean implies (Boolean b)**
  True if *self* is false, or if *self* is true and b is true. The rightmost implication is interpreted as an assertion of the right side of the expression. Further implications on the left side of an implication as well as implication in a Boolean expression are just evaluated to a Boolean value.
- **Boolean iff (Boolean b)**
  Shortcut for (a.implies(b) and b.implies(a)).
- **Boolean = (Boolean b)**
  Assigns the value *b* to the operand and returns *true*[11].

3.4.4 String

The standard type String represents strings, which can be ASCII.

- **Integer size ()**
  The number of characters in the *operand*.
- **String concat (String s)**
  The concatenation of the *operand* and *s*.
- **String substring (Integer lower, Integer upper)**

The sub-string of the *operand* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 0 to *size()*.

- **Boolean matches (String r)**
  Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-adress:
  ```
  mail.matches([\w]*@[\w]*.[\w]*);
  ```
- **Boolean substitutes (String r, String s)**
  Replaces all occurrences of the regular expression *r* in the *operand* by *s*. Regular expressions are given in the Java regular expression notation. For example, the following operation will substitute the occurrence of `"@"` with `"{at}"` in an e-mail-address:
  ```
  mail.substitutes("@", "{at}");
  ```
- **Integer toInteger ()**
  Converts the *operand* to an Integer value.
- **Real toReal ()**
  Converts the *operand* to a Real value.
- **Boolean = (String s)**
  Assigns the value *s* to the operand and returns *true*[11].

3.5 Enumeration Types

Enumerations allow the definition of sets of named values.

3.5.1 Enum

Enums inherit all operations from AnyType and adds the following operation:

- **Boolean = (Enum e)**
  Assigns the value *e* to the operand and returns *true*[11].

3.5.2 OrderedEnum

In contrast to Enums, individual ordinal values for the literals in an OrderedEnum are specified. Thus, an OrderedEnum defines a (total) ordering on its literals so that further operations in addition to those defined for Enum are available.

- **Boolean < (OrderedEnum l)**
  True if the *operand* is less than the ordinal value of the literal *l*.
- **Boolean > (OrderedEnum l)**
  True if the *operand* is greater than the ordinal value of the literal *l*.
- **Boolean <= (OrderedEnum l)**
  True if the *operand* is less than or equal to the ordinal value of the literal *l*.
- **Boolean >= (OrderedEnum l)**
  True if the *operand* is greater than or equal to the ordinal value of the literal *l*.

3.5.3 Constraint

The basic type Constraint represents variable constraints. In addition to the operations provided by AnyType, the Constraint type provides the following operations:

- **Boolean = (Constraint c)**
  Assigns the constraint *c* to the operandand returns *true*[11].

3.6 Collection Types

This section defines the operation of the collection types. The two IVML collections Set and Sequence are both subtypes of the abstract collection type Collection. Each collection type is actually a template type with one parameter. 'T' denotes the parameter. A concrete collection type is created by substituting a type for the T. So a collection of integers is referred in IVML by `setOf(Integer)`.

3.6.1 Collection

Collection is the abstract superclass of all collections in IVML.

- **Integer size ()**
  The number of elements in the collection *operand.*
- **Boolean includes (T object)**
  True if *object* is an element of *operand*, false otherwise.
- **Boolean excludes (T object)**
  True if *object* is not an element of *operand*, false otherwise.
- **Integer count (T object)**
  The number of times that *object* occurs in the collection *operand*.
- **Boolean isEmpty ()**
  Is the *operand* the empty collection?
- **Boolean notEmpty ()**
  Is the *operand* not the empty collection?
- **Boolean isDefined()**
  Returns whether (a variable of) the *operand* is defined, i.e. that an instance was already assigned.
- **T sum()**
  The addition of all elements in the *operand*. Elements must be of a type supporting the + operation (Integer or Real).
- **T product()**
  The multiplication of all elements in the *operand*. Elements must be of a type supporting the * operation (Integer or Real).
- **T min()**
  The minimum of all elements in the *operand*. Elements must be of a type supporting the < operation (Integer or Real).
- **T max()**
  The minimum of all elements in the *operand*. Elements must be of a type supporting the > operation (Integer or Real).
- **T avg()**

The average of all elements in the *operand*. Elements must be of a type supporting the / operation (Integer or Real).

- **Boolean forAll (Iterators | expression)**
  Results in true if *expression* evaluates to true for each element in the *operand* collection.
- **Boolean exists (Iterators | expression)**
  Results in true if *expression* evaluates to true for at least one element in the *operand* collection.
- **Boolean isUnique (Iterator | expression)**
  Results in true if *expression* evaluates to a different value for each element in the *operand* collection; otherwise, result is false. *isUnique* may have at most one iterator variable.
- **T any (Iterator | expression)**
  Returns any element in the *source* collection for which *expression* evaluates to true. If there is more than one element for which *expression* is true, one of them is returned. *any* may have at most one iterator variable.
- **Boolean one (Iterator | expression)**
  Results in true if there is exactly one element in the *operand* collection for which *expression* is true. *one* may have at most one iterator variable.
- **Collection<T> collect (Iterator | expression)**
  The Collection of elements that results from applying *expression* to every member of the *source* set. *collect* may have at most one iterator variable.
- **Collection<T> select (Iterator | expression)**
  The sub-collection for which expression is true. *select* may have at most one iterator variable.
- **Boolean reject (Iterator | expression)**
  The sub-collection for which expression is false. *reject* may have at most one iterator variable.
- **<R> apply (Iterator, R result | result = expression)**
  Applies the given expression to the operand collection using the specified iterator and stores the result in the last iterator (used here as a local variable declaration) which is returned as the result of this operation. Expression shall use the result “iterator” for aggregating values. Apply may have at most one iterator variable and needs to specify the result “iterator”.

3.6.2 Set

The Set is the mathematical set. It contains elements without duplicates. Set inherits the operations from Collection.

- **Boolean == (Set<T> s)**
  Evaluates to true if *operand* and *s* contain the same elements.
- **Set<T> union (Set<T> s)**
  The union of *operand* and *s*.
- **Set<T> intersection (Set<T> s)**
  The intersection of *operand* and *s* (i.e., the set of all elements that are in both *operand* and *s*).
- **Set<T> excluding (T object)**

The set containing all elements of *operand* without *object*.

- **Set<T> including (T object)**
  The set containing all elements of *operand* plus *object*.
- **Set<T> asSet ()**
  A Set identical to *operand*. This operation exists for convenience reasons.
- **Sequence<T> asSequence ()**
  A Sequence that contains all the elements from *operand,* in undefined order.
- **Set<T> typeSelect (MetaType T)**
  Results the subset of elements from *operand* which are of type *T*.
- **Set<T> typeReject (MetaType T)**
  Results the subset of elements from *operand* which are not of type *T*.
- **Boolean = (Set<T> s)**
  Assigns the value *s* to the operand and returns *true*[11].

3.6.3 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Sequence inherits the operations from Collection.

- **Boolean == (Sequence<T> s)**
  Evaluates to true if *operand* and *s* contain the same elements.
- **Sequence<T> union (Sequence<T> s)**
  The union of *operand* and *s*.
- **Set<T> asSet ()**
  The Set containing all the elements from *operand*, with duplicates removed.
- **Sequence<T> asSequence ()**
  The Sequence identical to the *operand* itself. This operation exists for convenience reasons.
- **T at (Integer i)**
  The *i-th* element of the sequence *operand*. Valid indices run from 0 to *size()-1*.
- **T [] (Integer i)**
  The *i-th* element of the sequence *operand*. This operation is an alias for at. Valid indices run from 0 to *size()-1*.
- **T first ()**
  The first element in *operand*.
- **T last()**
  The last element in *operand*.
- **Sequence<T> append (T object)**
  The sequence of elements, consisting of all elements of *operand*, followed by *object*.
- **Sequence<T> prepend(T object)**
  The sequence consisting of *object*, followed by all elements in *operand*.
- **Sequence<T> insertAt(Integer index, T object)**
  The sequence consisting of *operand* with *object* inserted at position *index*. Valid indices run from 0 to *size()-1*.
- **Integer indexOf(T object)**
  The index of object *object* in the sequence *operand*.

- **Sequence<T> typeSelect (MetaType T)**
  Results the subset of elements from *operand* which are of type *T*.
- **Sequence<T> typeReject (MetaType T)**
  Results the subset of elements from *operand* which are not of type *T*.
- **Boolean = (Sequence<T> s)**
  Assigns the value *s* to the operand and returns *true*[11].

3.7 Compound Types

A compound type groups multiple types into a single named unit. A compound inherits all its operations from AnyType. Access to variable declarations within a compound are specified using ".". Using the type name of the compound on the left side of a "." is a shortcut for an all-quantification on all instances of that compound. In addition, it defines the following operation:

- **Boolean isDefined()**
  Returns whether (a variable of) the *operand* is defined, i.e. that an instance was already assigned.
- **Boolean = (Compund c)**
  Assigns the value *c* to the operand and returns *true*[11].

4 Implementation Status

The realization of IVML and IVML-related tools is still in progress, In this section, we summarize the current status. We will first indicate the support for core IVML concepts in Table 1, then for advanced concepts in Table 2. Please note that DSL support in Table 2 was skipped already in early stages as it was not considered as relevant in the INDENICA project anymore.

IVML concept	IVML model, parser, translator	EASy reasoning support	EASy Configuration support
project	x	(x)	x
Boolean	x	x	x
integer	x	narrowing interval values	x
real	x	x	x
string	x	?	x
enumerations	x	x	x
container	x	?	x
type derivation and restriction	x	x	x
compounds	x	x	x
null values	x	not supported	x
decision variables	x	x	x
constraints	x	normal, internal (custom type constraints), compound constraints	-
constraints as variables	x	not supported	not supported
configurations	x	shall determine initial values	still determines initial values

Table 1: Implemented IVML core concepts (x=full support, -=no support as not responsible, partial support indicated by text)

IVML concept	IVML model, parser, translator	EASy reasoning support	EASy Configuration support
attributes	x	not supported	initial values
extended compounds	x	?	x
referenced elements	x	not supported	x
project versioning	x	-	-
project composition	resolution collides with editor	correct recursive processing missing	x (predecessor locations not passed to IVML)
project interfaces	x	-	x
partial configuration	x	x	x
freezing configurations	x	x	x
partial evaluation	x	correct recursive processing missing	?
DSL inclusion	(x)	-	-

Table 2: Implemented IVML advanced concepts (x=full support, -=no support as not responsible, partial support indicated by text)

5 IVML Grammar

In this section we depict the actual grammar for IVML. The grammar is given in six sections (basic modeling concepts, basic types and values, advanced modeling concepts, basic constraints, advanced constraints and terminals) in terms of a simplified xText[12] grammar (close to ANTLR[13] or EBNF). Simplified means, that we omitted technical details in xText used to properly generate the underlying EMF model as well as trailing ";" (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

5.1 Basic modeling concepts

```
VariabilityUnit:
      Project*

Project:
      'project' Identifier '{'
            VersionStmt?
            ImportStmt*
            ConflictStmt*
            InterfaceDeclaration*
            ProjectContents
      '}' ';'?

ProjectContents:
      (Typedef
            | VariableDeclaration
            | Freeze
            | Eval
            | ExpressionStatement
            | AttributeTo
            | OpDefStatement
            | AttrAssignment
      )*
```

[12] http://www.eclipse.org/Xtext/

[13] http://www.antlr.org

```
ExpressionBlock:
      '{'
            ExpressionStatement+
      '}' ';'?

Typedef:
      TypedefEnum
      | TypedefCompound
      | TypedefMapping

TypedefEnum:
      'enum' Identifier
      '{'
            TypedefEnumLiteral (',' TypedefEnumLiteral)*
      '}'
      TypedefConstraint?

TypedefEnumLiteral:
      Identifier ('=' NumValue)?

TypedefCompound:
      'compound' Identifier ('refines' Identifier)?
      '{'
            (VariableDeclaration
                  | ExpressionStatement
                  | AttrAssignment)*
      '}' ';'?

TypedefMapping:
      'typedef' Identifier Type TypedefConstraint? ';'

TypedefConstraint:
      'with' '('Expression (',' Expression)* ')'

VariableDeclaration:
      Type VariableDeclarationPart (',' VariableDeclarationPart)* ';'

VariableDeclarationPart:
      Identifier ('=' Expression)?
```

```
DerivedType:
      (
            'setOf'
            | 'sequenceOf'
            | 'refTo'
      )
      '(' Type ')'
```

5.2 Basic types and values

```
BasicType:
      'Integer'
      | 'Real'
      | 'Boolean'
      | 'String'
      | 'Constraint'

Type:
      BasicType
      | QualifiedName
      | DerivedType

NumValue:
      NUMBER

QualifiedName:
      (Identifier '::' (Identifier '::')*)? Identifier

AccessName:
      ('.' Identifier)+

Value:
      NumValue
      | STRING
      | QualifiedName
      | ('true' | 'false')
```

5.3 Advanced modeling concepts

```
AttributeTo :
      'attribute' Type VariableDeclarationPart 'to' Identifier
      (',' Identifier)*';'

AttrAssignment:
    'assign'
    '(' AttrAssignmentPart (',' AttrAssignmentPart)* ')' 'to'
    '{'
       (VariableDeclaration | ExpressionStatement | AttrAssignment)+
    '}' ';'?

AttrAssignmentPart:
    Identifier '=' LogicalExpression

Freeze:
      'freeze' '{'
            FreezeStatement+
      '}' ('but' FreezeButList)? ';'?

FreezeStatement:
      QualifiedName AccessName? ';'

FreezeButList:
      '(' FreezeButExpression (',' FreezeButExpression)* ')'

FreezeButExpression:
      QualifiedName AccessName? '*'?

Eval:
      'eval' ExpressionBlock

InterfaceDeclaration:
      'interface' Identifier '{'
            Export*
      '}' ';'?

Export:
      'export' Identifier (',' Identifier)* ';'
ImportStmt:
      'import' Identifier ('::' Identifier)?
```

```
	(
		'with' '(' VersionedId (',' VersionedId)*')'
	)? ';'

ConflictStmt:
	'conflicts' Identifier
	(
		'with' '(' VersionedId (',' VersionedId)* ')'
	)? ';'

VersionedId:
	Identifier '.version' VersionOperator VERSION

VersionOperator:
	'==' | '>' | '<' | '>=' | '<=' | '<>' | '!='

VersionStmt:
	'version' VERSION ';'

DslContext:
	'DSL' '(' STRING ',' STRING ',' STRING ')'
	DSL_CONTENT
```

5.4 Basic constraints

```
ExpressionStatement:
	Expression ';'


Expression:
	LetExpression
	| ImplicationExpression
	| CollectionInitializer
	| DslContext



ImplicationExpression:
	AssignmentExpression ImplicationExpressionPart*
```

```
ImplicationExpressionPart:
      ImplicationOperator AssignmentExpression

ImplicationOperator:
      'implies' | 'iff'

AssignmentExpression:
    LogicalExpression AssignmentExpressionPart?

AssignmentExpressionPart:
    '=' (LogicalExpression | CollectionInitializer)

LogicalExpression:
      EqualityExpression LogicalExpressionPart*

LogicalExpressionPart:
      LogicalOperator EqualityExpression

LogicalOperator:
      'and' | 'or' | 'xor'

EqualityExpression:
    RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:
    EqualityOperator (RelationalExpression | CollectionInitializer)

EqualityOperator:
    '==' | '<>' | '!='

RelationalExpression:
      AdditiveExpression RelationalExpressionPart?

RelationalExpressionPart:
      RelationalOperator AdditiveExpression



RelationalOperator:
      '>' | '<' | '>=' | '<=' | '<>' | '!='
```

```
AdditiveExpression:
      MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:
      AdditiveOperator MultiplicativeExpression

AdditiveOperator:
      '+' | '-'

MultiplicativeExpression:
      UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:
      MultiplicativeOperator UnaryExpression

MultiplicativeOperator:
      '*' | '/'

UnaryExpression:
      UnaryOperator? PostfixExpression

UnaryOperator:
      'not' | '-'

PostfixExpression:
      (FeatureCall Call* ExpressionAccess?)
      | PrimaryExpression

Call:
      '.' FeatureCall
      | '->' SetOp
      | '[' Expression ']'

FeatureCall:
      Identifier '(' ActualParameterList? ')'


SetOp:
      Identifier
```

```
    '(' Declarator Expression? ')'

Declarator:
    Declaration (';' Declaration)* '|'

Declaration:
    Identifier (',' Identifier)* (':' Type)? ('=' Expression)?

ActualParameterList:
    Expression (',' Expression)*

ExpressionAccess:
    '.' Identifier Call* ExpressionAccess?

PrimaryExpression:
    (
        Literal
        | '(' Expression ')'
        | IfExpression
        | 'refBy' '(' Identifier ')'
    )
    Call*
    ExpressionAccess?

CollectionInitializer:
    QualifiedName?
    '{'
        ExpressionList?
    '}'

ExpressionList:
    ExpressionListEntry (',' ExpressionListEntry)*

ExpressionListEntry:
    (Identifier ('.' Identifier)? '=')?
    (LogicalExpression | LiteralCollection)


Literal:
    Value
```

5.5 Advanced constraints

```
LetExpression:
      'let' Type Identifier '=' Expression 'in' Expression

IfExpression:
      'if' Expression 'then' Expression 'else' Expression 'endif'

OpDefStatement:
      'def' Type Identifier '(' OpDefParameterList ')'
      '=' Expression ';'

OpDefParameterList:
      (OpDefParameter (',' OpDefParameter)* )?

OpDefParameter:
      Type Identifier ('=' Expression)?
```

5.6 Terminals

```
Identifier:
    ID | VERSION | EXPONENT

terminal VERSION:
      'v' ('0'..'9')+ ('.' ('0'..'9')+)*

terminal ID:
      ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*

terminal NUMBER:
      '-'?
      (('0'..'9')+ ('.' ('0'..'9')* EXPONENT?)?
      | '.' ('0'..'9')+ EXPONENT?
      | ('0'..'9')+ EXPONENT)

terminal EXPONENT:
      ('e'|'E') ('+'|'-')? ('0'..'9')+
terminal STRING   :
      '"' (
```

```
            '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"')
      )* '"'
      | "'" (
            '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'")
      )* "'"

terminal DSL_CONTENT:
      '%' -> 'DSL%'

terminal ML_COMMENT:
      '/*' -> '*/'

terminal SL_COMMENT:
      '//' !('\n'|'\r')* ('\r'? '\n')?

terminal WS:
      (' '|'\t'|'\r'|'\n')+

terminal ANY_OTHER:
      .
```

References

[1] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2010.

[2] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variabilits Modelling of Software-Intensive Systems (VaMoS '10)*, pages 159–162, 2010.

[3] E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors. *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. IEEE, 2011.

[4] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[5] Object Management Group, Inc. (OMG). Unified Modeling Language: Superstructure version 2.1.2. Specification v2.11 2007-11-02, Object Management Group, November 2007. Available online at: http://www.omg.org/docs/formal/2007-11-02.pdf.

[6] Object Management Group, Inc. (OMG). Common Variability Language (CVL), 2010. OMG initial submission. Available on request.

[7] Mark-Oliver Reiser. Core Concepts of the Compositional Variability Management Framework (CVM). Technical Report 2009/16, Technische Universität Berlin, 2009. Available online at http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/.

[8] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '10)*, pages 123–130, 2010.

[9] Rob van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.